Distributed Computing and Systems
Chalmers university of technology

Lock-Free and Practical Doubly Linked List-Based Deques using Single-Word Compare-And-Swap

Håkan Sundell

Philippas Tsigas

OPODIS 2004:
The 8th International Conference on Principles of Distributed Systems

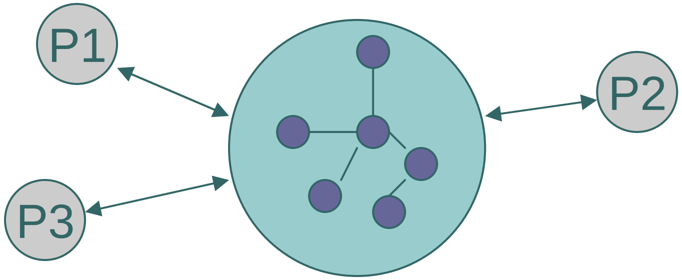
Sundell Jr.

Outline

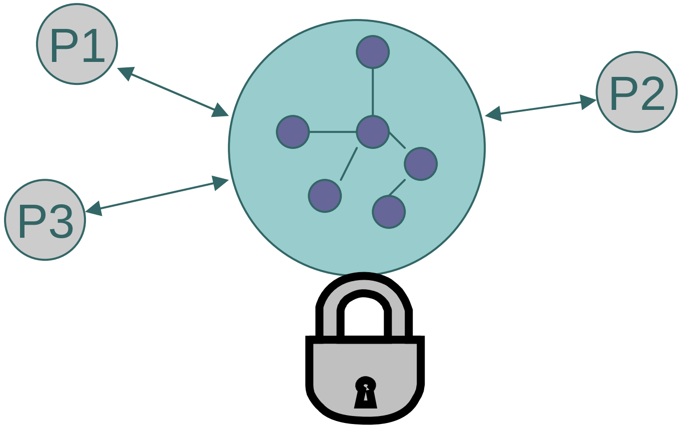
- Synchronization Methods
- Deques (Double-Ended Queues)
- Doubly Linked Lists
- Concurrent Deques
  - Previous results
  - New Lock-Free Algorithm
- Experimental Evaluation
- Conclusions

Synchronization

- Shared data structures needs synchronization



- Synchronization using Locks
  - Mutually exclusive access to whole or parts of the data structure



Blocking Synchronization

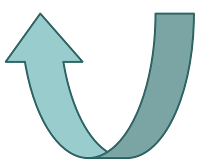
- Drawbacks
  - Blocking
  - Priority Inversion
  - Risk of deadlock

- Locks: Semaphores, spinning, disabling interrupts etc.
  - Reduced efficiency because of reduced parallelism

Non-blocking Synchronization

- Lock-Free Synchronization
  - Optimistic approach (i.e. assumes no interference)

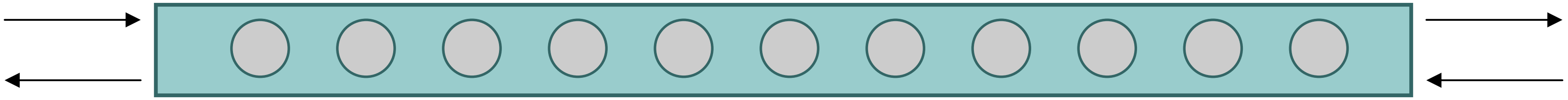
    1. The operation is prepared to later take effect (unless interfered) using hardware atomic primitives



    2. Possible interference is detected via the atomic primitives, and causes a retry
       - Can cause starvation
- Wait-Free Synchronization
  - Always finishes in a finite number of its own steps.

Deques (Double-Ended Queues)

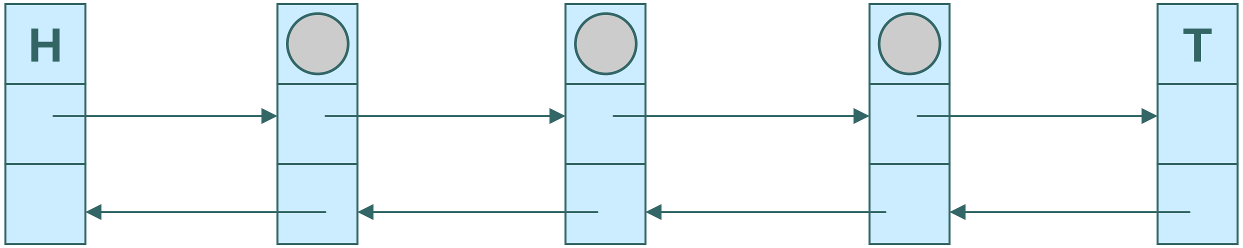
- Fundamental data structure
- Stores values that can be removed depending on the store order.
  - Incorporates the functionality of both stacks and queues



- Four basic operations:
  - PushRight/Left(v): Adds a new item
  - v=PopRight/Left(): Removes an item

Doubly Linked Lists

- Fundamental data structure
  - Can be used to implement various abstract data types (e.g. deques)



- Unordered List, i.e. the nodes are ordered only relatively to each other.
- Supports Traversals
- Supports Inserts/Deletes at arbitrary positions

Previous Non-blocking Deques (Doubly Linked Lists)

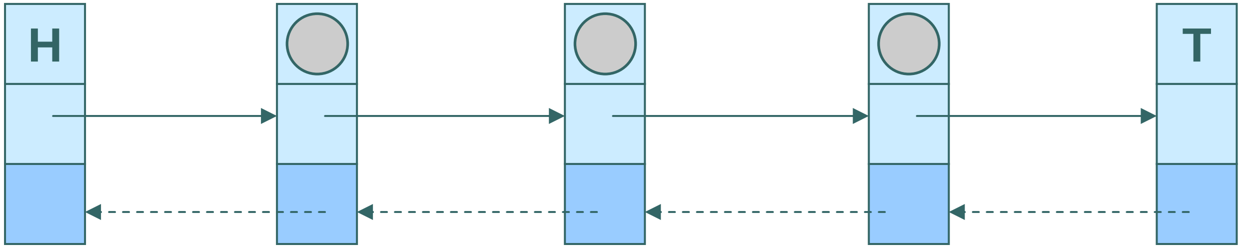
- M. Greenwald, “Two-handed emulation: how to build non-blocking implementations of complex data structures using DCAS”, PODC 2002
- O. Agesen et al., “DCAS-based concurrent deques”, SPAA 2000
  - D. Detlefs et al., “Even better DCAS-based concurrent deques”, DISC 2000
  - P. Martin et al. “DCAS-based concurrent deques supporting bulk allocation”, TR, 2002
  - *Errata:* S. Doherty et al. “DCAS is not a silver bullet for nonblocking algorithm design”, SPAA 2004

Previous Non-blocking Deques

- N. Arora et al., “Thread scheduling for multiprogrammed multiprocessors”, SPAA 1998
  - Not full deque semantics
  - Limited concurrency
- M. Michael, “CAS-based lock-free algorithm for shared deques”, EuroPar 2003
  - Requires double-width CAS
  - Not disjoint-access-parallel

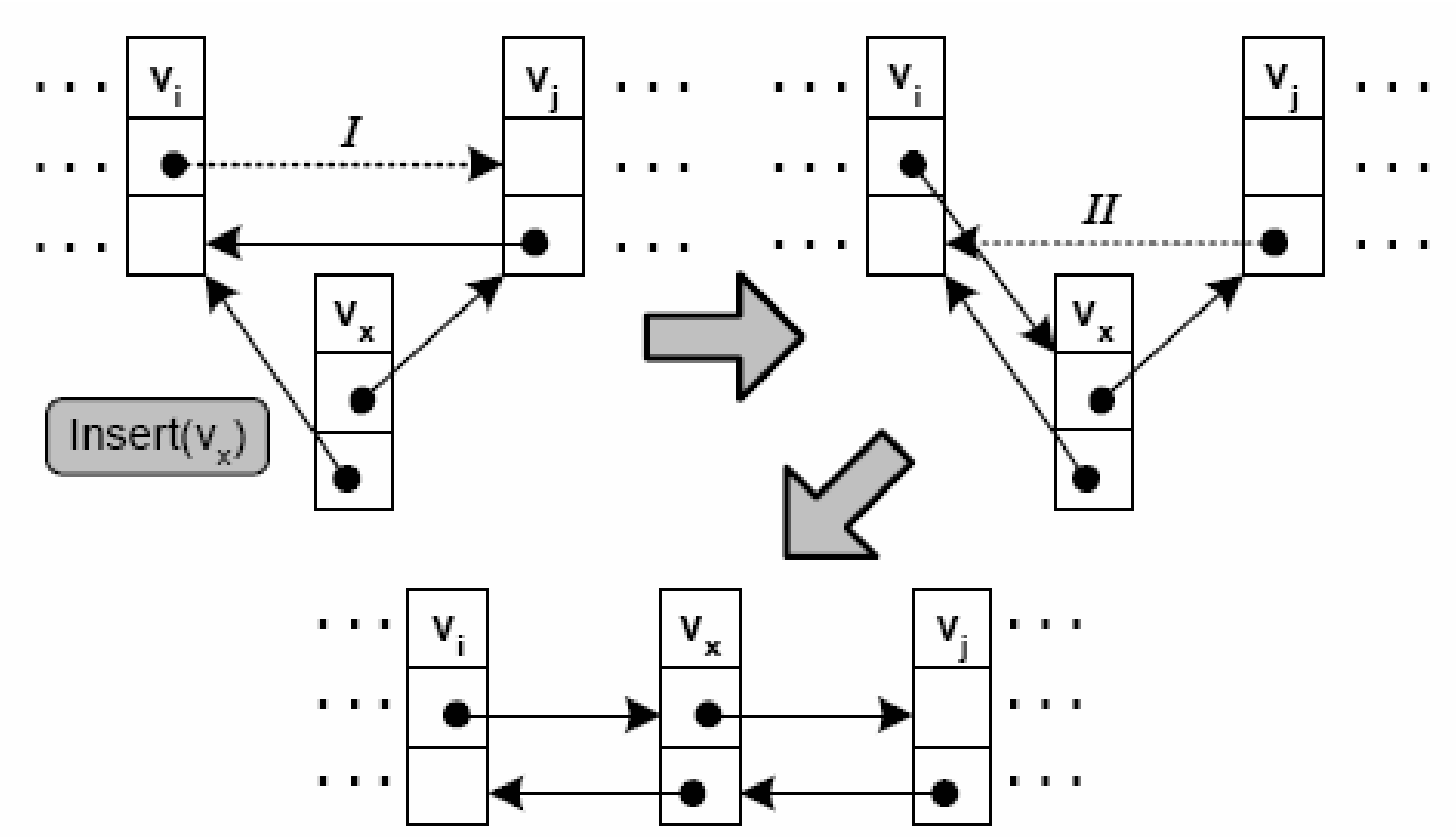
New Lock-Free Concurrent Doubly Linked List

- Treat the doubly linked list as a singly linked list with auxiliary information in each node about its predecessor!

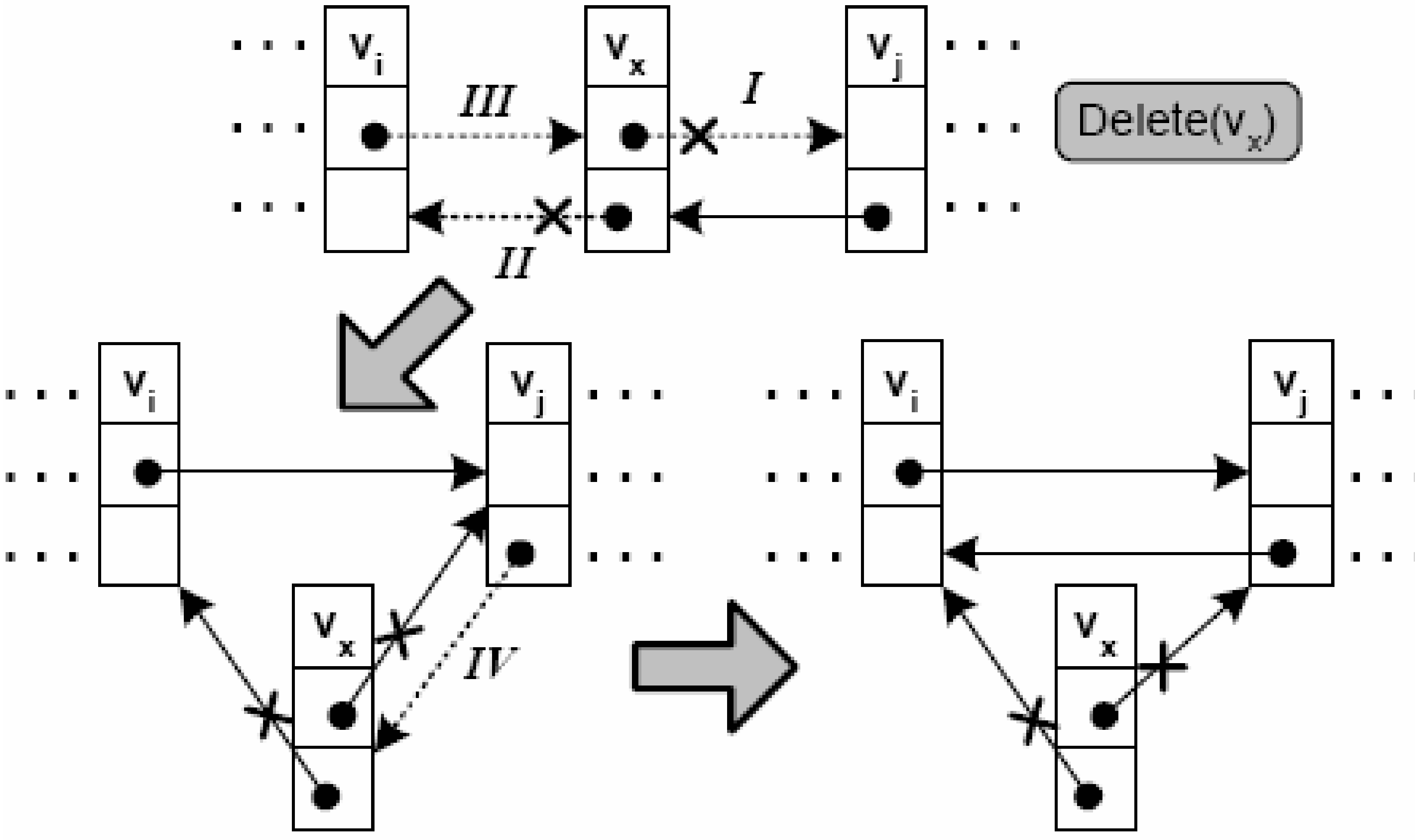


- Singly Linked Lists
  - T. Harris, “A pragmatic implementation of non-blocking linked lists”, DISC 2001
    - Marks pointers using spare bit
    - Needs only standard CAS

Lock-Free Doubly Linked Lists - INSERT



Lock-Free Doubly Linked Lists - DELETE



Lock-Free Doubly Linked List - Memory Management

- The information about neighbor nodes should also be accessible in partially deleted nodes!
  - Enables helping operations to find
  - Enables continuous traversals
- M. Michael, “Safe memory reclamation for dynamic lock-free objects using atomic read and writes”, PODC 2002
  - Does not allow pointers from nodes

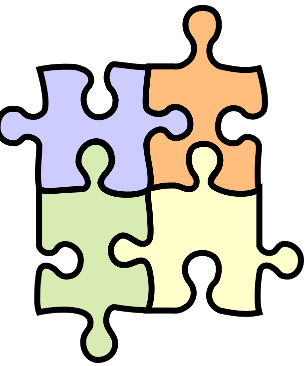
Lock-Free Doubly Linked List - Memory Management

- D. Detlefs et al., “Lock-Free Reference Counting”, PODC 2001
  - Uses DCAS, which is not available
- J. Valois, “Lock-Free Data Structures”, 1995
  - M. Michael and M. Scott, “Correction of a memory management method for lock-free data structures”, 1995
    - Uses standard CAS
    - Uses free-list style of memory pool

Lock-Free Doubly Linked List - Cyclic Garbage Avoidance

- Lock-Free Reference Counting is sufficient for our algorithm.
- Reference Counting can not handle cyclic garbage!
  - We break the symmetry directly before possible reclaiming a node, such that helping operations still can utilize the information in the node.
  - We make sure that next and prev pointers from a deleted node, only points to active nodes.

New Lock-Free Doubly Linked List - Techniques Summary

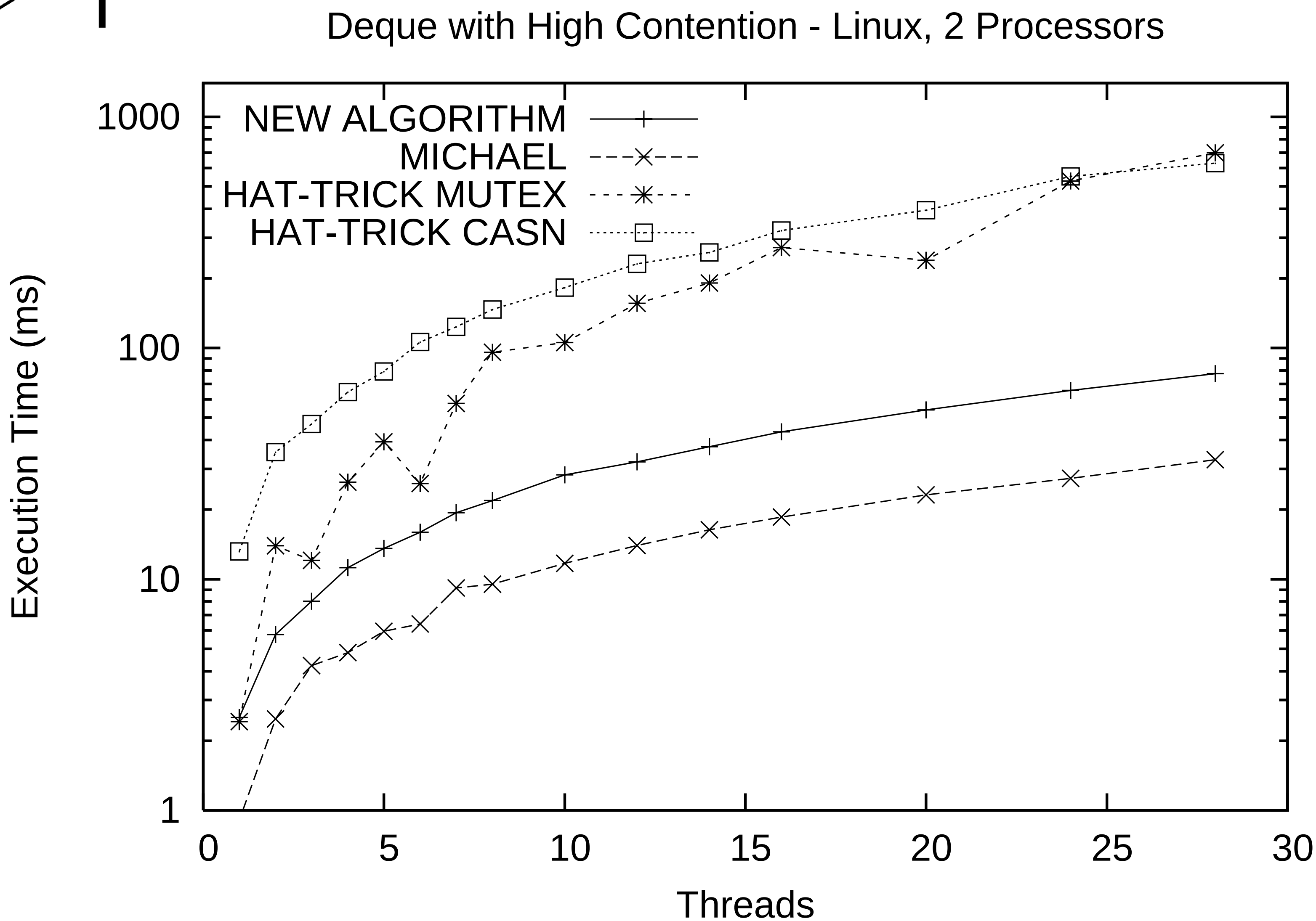


- General Doubly Linked List Structure
  - Treated as singly linked lists with extra info
- Uses CAS atomic primitive
- Lock-Free memory management
  - IBM Freelists
  - Reference counting (Valois+Michael&Scott)
- Avoids cyclic garbage
- Helping scheme
- All together proved to be linearizable

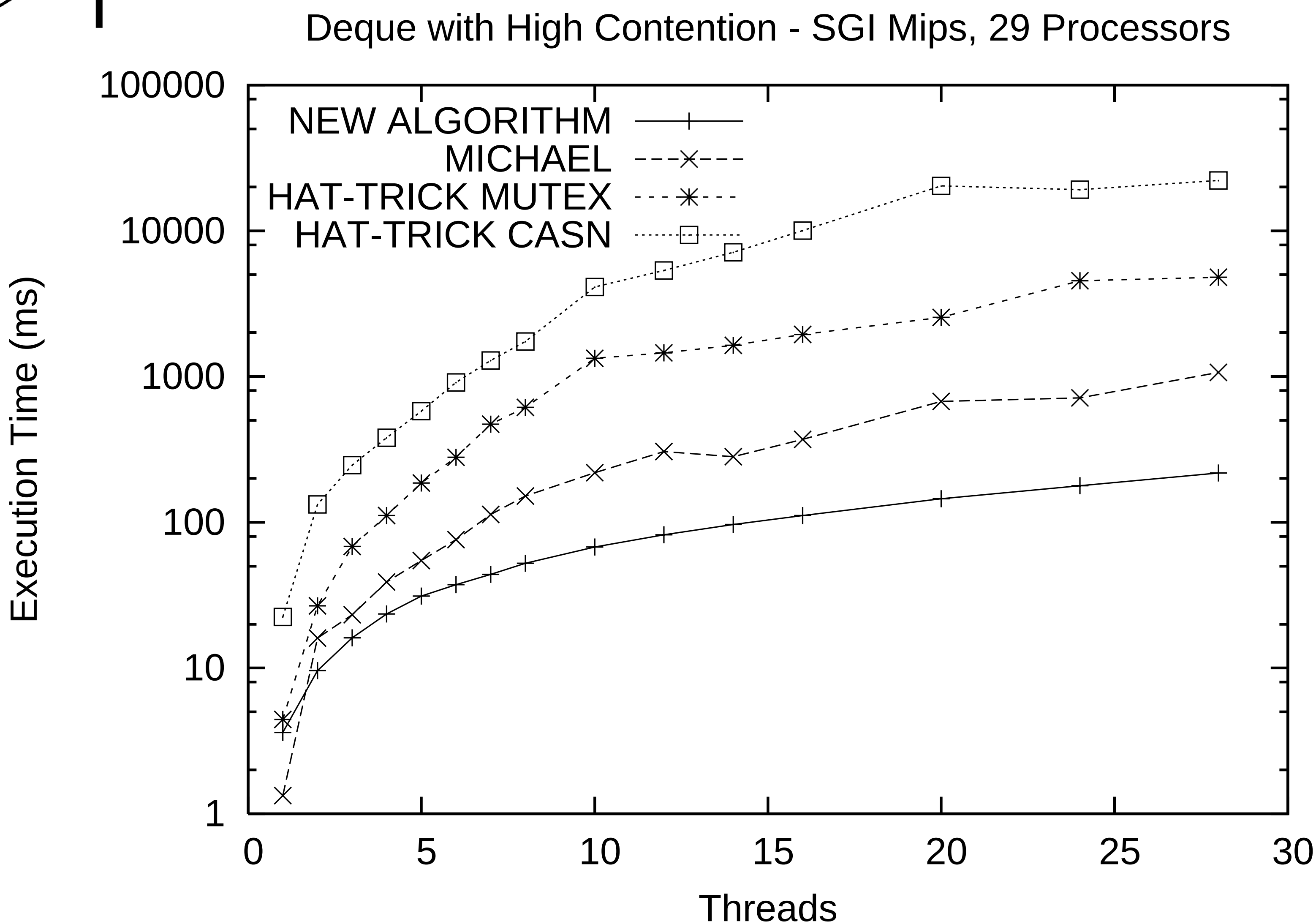
Experimental Evaluation

- Experiment with 1-28 threads performed on systems with 2, 4 respective 29 cpu's.
  - Each thread performs 1000 operations, randomly distributed over PushRight, PushLeft, PopRight and PopLeft's.
- Compare with implementation by Michael and Martin et al., using same scenarios.
- For Martin et al. DCAS implemented by software CASN by Harris et al. or by mutex.
- Averaged execution time of 50 experiments.

Linux Pentium II, 2 cpu's



SGI Origin 2000, 29 cpu's.



Conclusions

- A first lock-free Deque using single word CAS.
- The new algorithm is more scalable than Michael's, because of its disjoint-access-parallel property.
- Also implements a general doubly linked list, the first using CAS.
- Our lock-free algorithm is suitable for both pre-emptive as well as systems with full concurrency.
  - Will be available as part of NOBLE software library, http://www.noble-library.org
- See Håkan Sundell's PhD Thesis for an extended version of the paper.

Questions?

- Contact Information:

  - Address:

    Håkan Sundell or Philippas Tsigas
    Computing Science
    Chalmers University of Technology
  - Email:

    <phs , tsigas> @ cs.chalmers.se
  - Web:

    http://www.cs.chalmers.se/~noble

Lock-Free Doubly Linked Lists

```
union Link
        _: word
        ⟨p, d⟩: ⟨pointer to Node, boolean⟩

structure Node
        value: pointer to word
        prev: union Link
        next: union Link

// Global variables
head, tail: pointer to Node
```

Lock-Free Doubly Linked Lists

```
procedure PushLeft(value: pointer to word)
    node:=CreateNode(value);
    prev:=COPY_NODE(head);
    next:=READ_NODE(&prev.next);
    while true do
        if prev.next ≠ ⟨next,false⟩ then
            RELEASE_NODE(next);
            next:=READ_NODE(&prev.next);
            continue;
        node.prev:=⟨prev,false⟩;
        node.next:=⟨next,false⟩;
        if CAS(&prev.next,⟨next,false⟩,⟨node,false⟩) then
            COPY_NODE(node);
            break;
        Back-Off
    PushCommon(node,next);
```

Lock-Free Doubly Linked Lists

```
procedure PushRight(value: pointer to word)
    node:=CreateNode(value);
    next:=COPY_NODE(tail);
    prev:=READ_NODE(&next.prev);
    while true do
        if prev.next ≠ ⟨next,false⟩ then
            prev:=HelpInsert(prev,next);
            continue;
        node.prev:=⟨prev,false⟩;
        node.next:=⟨next,false⟩;
        if CAS(&prev.next,⟨next,false⟩,⟨node,false⟩) then
            COPY_NODE(node);
            break;
        Back-Off
    PushCommon(node,next);
```

Lock-Free Doubly Linked Lists

```
procedure PushCommon(node, next: pointer to Node)
    while true do
        link1:=next.prev;
        if link1.d = true or node.next ≠ ⟨next,false⟩ then
            break;
        if CAS(&next.prev,link1,⟨node,false⟩) then
            COPY_NODE(node);
            RELEASE_NODE(link1.p);
            if node.prev.d = true then
                prev2:=COPY_NODE(node);
                prev2:=HelpInsert(prev2,next);
                RELEASE_NODE(prev2);
            break;
        Back-Off
    RELEASE_NODE(next);
    RELEASE_NODE(node);
```

Lock-Free Doubly Linked Lists

```
function PopLeft(): pointer to word
PL1     prev:=COPY_NODE(head);
PL2     while true do
PL3         node:=READ_NODE(&prev.next);
PL4         if node = tail then
PL5             RELEASE_NODE(node);
PL6             RELEASE_NODE(prev);
PL7             return ⊥;
PL8         link1:=node.next;
PL9         if link1.d = true then
PL10            HelpDelete(node);
PL11            RELEASE_NODE(node);
PL12            continue;
```

Lock-Free Doubly Linked Lists

```
        if CAS(&node.next,link1,⟨link1.p,true⟩) then
            HelpDelete(node);
            next:=READ_DEL_NODE(&node.next);
            prev:=HelpInsert(prev,next);
            RELEASE_NODE(prev);
            RELEASE_NODE(next);
            value:=node.value;
            break;
        RELEASE_NODE(node);
        Back-Off
    RemoveCrossReference(node);
    RELEASE_NODE(node);
    return value;
```

Lock-Free Doubly Linked Lists

```
function PopRight(): pointer to word
    next:=COPY_NODE(tail);
    node:=READ_NODE(&next.prev);
    while true do
        if node.next ≠ ⟨next,false⟩ then
            node:=HelpInsert(node,next);
            continue;
        if node = head then
            RELEASE_NODE(node);
            RELEASE_NODE(next);
            return ⊥;
```

Lock-Free Doubly Linked Lists

```
    if CAS(&node.next,⟨next,false⟩,⟨next,true⟩) then
        HelpDelete(node);
        prev:=READ_DEL_NODE(&node.prev);
        prev:=HelpInsert(prev,next);
        RELEASE_NODE(prev);
        RELEASE_NODE(next);
        value:=node.value;
        break;
    Back-Off
RemoveCrossReference(node);
RELEASE_NODE(node);
return value;
```

Lock-Free Doubly Linked Lists

```
procedure HelpDelete(node: pointer to Node)
    while true do
        link1:=node.prev;
        if link1.d = true or
          CAS(&node.prev,link1,⟨link1.p,true⟩) then break;
    lastlink.d:=true;
    prev:=READ_DEL_NODE(&node.prev);
    next:=READ_DEL_NODE(&node.next);
    while true do
        if prev = next then break;
        if next.next.d = true then
            next2:=READ_DEL_NODE(&next.next);
            RELEASE_NODE(next);
            next:=next2;
            continue;
```

Lock-Free Doubly Linked Lists

```
prev2:=READ_NODE(&prev.next);
if prev2 = NULL then
    if lastlink.d = false then
        HelpDelete(prev);
        lastlink.d:=true;
    prev2:=READ_DEL_NODE(&prev.prev);
    RELEASE_NODE(prev);
    prev:=prev2;
    continue;
if prev2 ≠ node then
    lastlink.d:=false;
    RELEASE_NODE(prev);
    prev:=prev2;
    continue;
```

Lock-Free Doubly Linked Lists

```
    RELEASE_NODE(prev2);
    if CAS(&prev.next,⟨node,false⟩,⟨next,false⟩) then
        COPY_NODE(next);
        RELEASE_NODE(node);
        break;
    Back-Off
RELEASE_NODE(prev);
RELEASE_NODE(next);
```

Lock-Free Doubly Linked Lists

```
function HelpInsert(prev, node: pointer to Node)
  :pointer to Node
        lastlink.d:=true;
        while true do
            prev2:=READ_NODE(&prev.next);
            if prev2 = NULL then
                if lastlink.d = false then
                    HelpDelete(prev);
                    lastlink.d:=true;
                prev2:=READ_DEL_NODE(&prev.prev);
                RELEASE_NODE(prev);
                prev:=prev2;
                continue;
```

Lock-Free Doubly Linked Lists

```
    link1:=node.prev;
    if link1.d = true then
        RELEASE_NODE(prev2);
        break;
    if prev2 ≠ node then
        lastlink.d:=false;
        RELEASE_NODE(prev);
        prev:=prev2;
        continue;
    RELEASE_NODE(prev2);
    if CAS(&node.prev,link1,⟨prev,false⟩) then
        COPY_NODE(prev);
        RELEASE_NODE(link1.p);
        if prev.prev.d = true then continue;
        break;
    Back-Off
return prev;
```

Lock-Free Doubly Linked Lists

```
procedure RemoveCrossReference(node: pointer to Node)
    while true do
        prev:=node.prev.p;
        if prev.next.d = true then
            prev2:=READ_DEL_NODE(&prev.prev);
            node.prev:=⟨prev2,true⟩;
            RELEASE_NODE(prev);
            continue;
        next:=node.next.p;
        if next.next.d = true then
            next2:=READ_DEL_NODE(&next.next);
            node.next:=⟨next2,true⟩;
            RELEASE_NODE(next);
            continue;
        break;
```

Lock-Free Doubly Linked Lists

- Is really PopLeft linarizable?
  - We can not guarantee that the node is the first, at the same time as we logically delete it!
  - No problem: we can safely assume that the node was deleted at the time we verified that the node was the first, as this operation was the only one to delete it and no other operation cares about the deletion state of that node for its result.

Lock-Free Doubly Linked Lists

- How can we traverse through nodes that are logically (and maybe even ”physically”) deleted?
  - We interpret the ”cursor” position as the node itself, or if its get deleted, the position will be inherited to its next node (interpreted as directly before that one)
    - Applied recursively, if next node is also deleted

Lock-Free Doubly Linked Lists

```
function Next(cursor: pointer to pointer to Node): boolean
NT1     while true do
NT2         if *cursor = tail then return false;
NT3         next:=READ_DEL_NODE(&(*cursor).next);
NT4         d := next.next.d;
NT5         if d = true and (*cursor).next ≠ ⟨next,true⟩ then
NT6             if (*cursor).next.p = next then HelpDelete(next);
NT7             RELEASE_NODE(next);
NT8             continue;
NT9         RELEASE_NODE(*cursor);
NT10        *cursor:=next;
NT11        if d = false and next ≠ tail then return true;
```

Lock-Free Doubly Linked Lists

```
function Prev(cursor: pointer to pointer to Node): boolean
PV1     while true do
PV2         if *cursor = head then return false;
PV3         prev:=READ_DEL_NODE(&(*cursor).prev);
PV4         if prev.next = ⟨*cursor,false⟩ and (*cursor).next.d = false then
PV5             RELEASE_NODE(*cursor);
PV6             *cursor:=prev;
PV7             if prev ≠ head then return true;
PV8         else if (*cursor).next.d = true then Next(cursor);
PV9         else
PV10            prev:=HelpInsert(prev,*cursor);
PV11            RELEASE_NODE(prev);
```

Lock-Free Doubly Linked Lists

```
procedure InsertBefore(cursor: pointer to pointer to Node,
  value: pointer to word)
IB1     if *cursor = head then return InsertAfter(cursor,value);
IB2     node:=CreateNode(value);
IB3     while true do
IB4         if (*cursor).next.d = true then Next(cursor);
IB5         prev:=READ_DEL_NODE(&(*cursor).prev);
IB6         node.prev:=⟨prev,false⟩;
IB7         node.next:=⟨(*cursor),false⟩;
IB8         if CAS(&prev.next,⟨(*cursor),false⟩,⟨node,false⟩) then
IB9             COPY_NODE(node);
IB10            break;
IB11        if prev.next ≠ ⟨(*cursor),false⟩ then prev:=HelpInsert(prev,*cursor);
IB12        RELEASE_NODE(prev);
IB13        Back-Off
IB14    next:=(*cursor);
IB15    *cursor:=COPY_NODE(node);
IB16    node:=HelpInsert(node,next);
IB17    RELEASE_NODE(node);
IB18    RELEASE_NODE(next);
```

Lock-Free Doubly Linked Lists

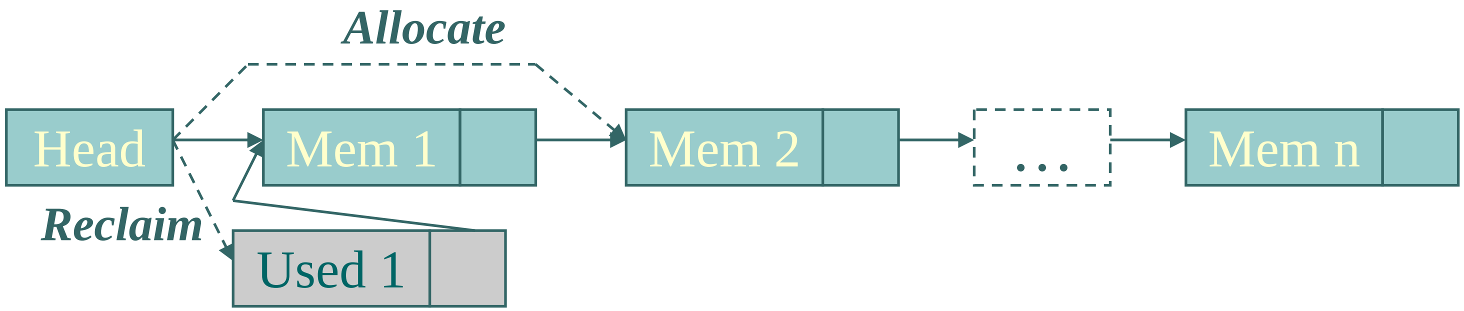
```
procedure InsertAfter(cursor: pointer to pointer to Node,
 value: pointer to word)
IA1     if *cursor = tail then return InsertBefore(cursor,value);
IA2     node:=CreateNode(value);
IA3     while true do
IA4        next:=READ_DEL_NODE(&(*cursor).next);
IA5        node.prev:=⟨(*cursor),false⟩;
IA6        node.next:=⟨next,false⟩;
IA7        if CAS(&(*cursor).next,⟨next,false⟩,⟨node,false⟩) then
IA8           COPY_NODE(node);
IA9           break;
IA10       RELEASE_NODE(next);
IA11       if (*cursor).next.d = true then
IA12          RELEASE_NODE(node);
IA13          return InsertBefore(cursor,value);
IA14       Back-Off
IA15    *cursor:=COPY_NODE(node);
IA16    node:=HelpInsert(node,next);
IA17    RELEASE_NODE(node);
IA18    RELEASE_NODE(next);
```

Lock-Free Doubly Linked Lists

```
function Delete(cursor: pointer to pointer to Node): point
    if *cursor = head or *cursor = tail then return ⊥;
    while true do
        link1:=(*cursor).next;
        if link1.d = true then return ⊥;
        if CAS(&(*cursor).next,link1,⟨link1.p,true⟩) then
            HelpDelete(*cursor);
            prev:=COPY_NODE((*cursor).prev.p);
            prev:=HelpInsert(prev,link1.p);
            RELEASE_NODE(prev);
            value:=(*cursor).value;
            RemoveCrossReference(*cursor);
            return value;
```

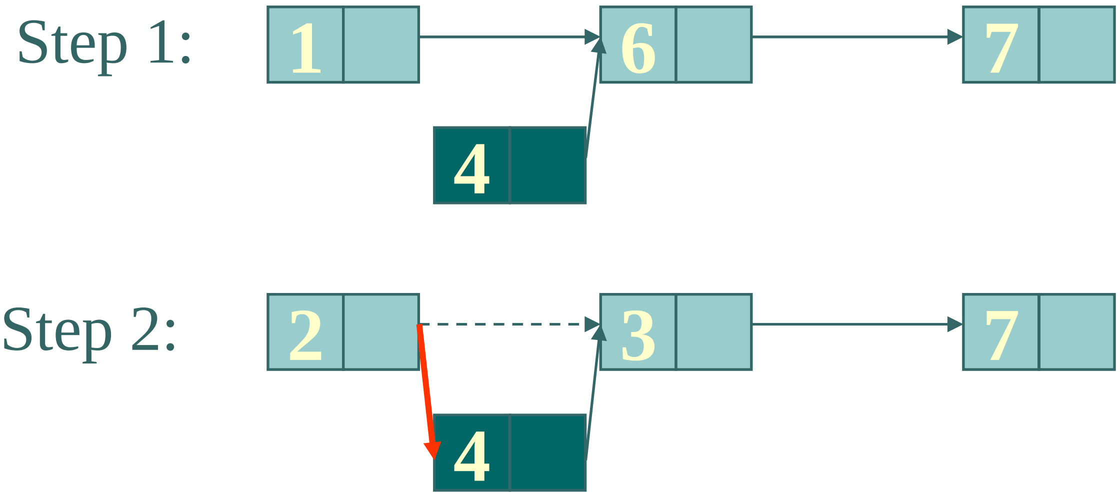
Dynamic Memory Management

- Problem: System memory allocation functionality is blocking!
- Solution (lock-free), IBM freelists:
  - Pre-allocate a number of nodes, link them into a dynamic stack structure, and allocate/reclaim using CAS



The ABA problem

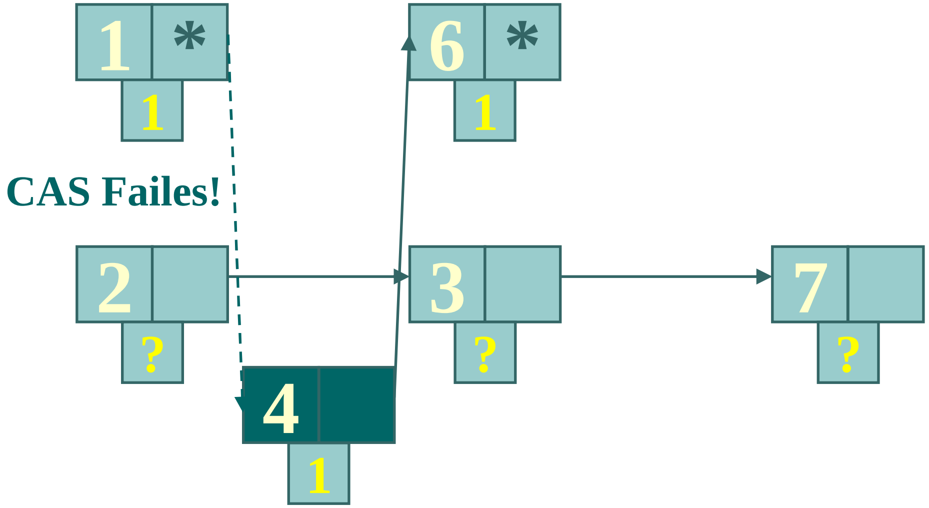
- Problem: Because of concurrency (pre-emption in particular), same pointer value does not always mean same node (i.e. CAS succeeds)!!!



The ABA problem

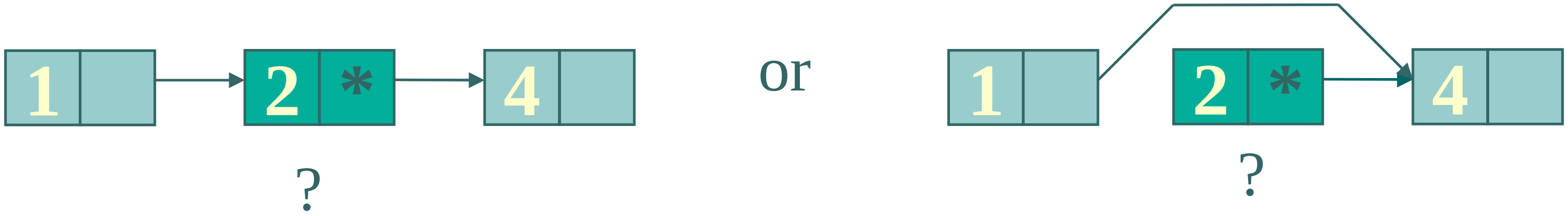
- Solution: (Valois et al) Add reference counting to each node, in order to prevent nodes that are of interest to some thread to be reclaimed until all threads have left the node

New Step 2:



Helping Scheme

- Threads need to traverse safely



- Need to remove marked-to-be-deleted nodes while traversing – Help!
- Finds previous node, finish deletion and continues traversing from previous node

Back-Off Strategy

- For pre-emptive systems, helping is necessary for efficiency and lock-freeness
- For really concurrent systems, overlapping CAS operations (caused by helping and others) on the same node can cause heavy contention
- Solution: For every failed CAS attempt, back-off (i.e. sleep) for a certain duration, which increases exponentially

Non-blocking Synchronization

- Lock-Free Synchronization
  - Avoids problems with locks
  - Simple algorithms
  - Fast when having low contention
- Wait-Free Synchronization
  - Always finishes in a finite number of its own steps.
    - Complex algorithms
    - Memory consuming
    - Less efficient in average than lock-free

Correctness

- Linearizability (Herlihy 1991)
  - In order for an implementation to be ***linearizable***, for every concurrent execution, there should exist an ***equal sequential execution*** that respects the ***partial order*** of the operations in the concurrent execution

Correctness

- Define precise sequential semantics
- Define abstract state and its interpretation
  - Show that state is atomically updated
- Define linearizability points
  - Show that operations take effect atomically at these points with respect to sequential semantics
- Creates a total order using the linearizability points that respects the partial order
  - The algorithm is linearizable

Correctness

- Lock-freeness
  - At least one operation should always make progress
- There are no cyclic loop depencies, and all potentially unbounded loops are ”gate-keeped” by CAS operations
  - The CAS operation guarantees that at least one CAS will always succeed
    - The algorithm is lock-free